Rapport interne **IRIT-97-09-R**

Prédiction de l'adresse des lectures pour tolérer la latence des accès mémoire

Prediction of addresses for speculating loads

Pham Tuong Hai, Christine Rochange, Pascal Sainrat, Daniel Litaize

Résumé

Un des points clés de l'amélioration de la performance des processeurs superscalaires est la tolérance de la latence des accès aux données en mémoire. Nous proposons de résoudre ce problème en prédisant, lors de l'émission dans une station de réservation d'une instruction de lecture, l'adresse à laquelle elle va accéder. Ainsi, les instructions de lecture peuvent être exécutées plus tôt, masquant ainsi tout ou partie de la latence de la hiérarchie mémoire. La stratégie de prédiction de l'adresse ressemble à celle utilisée par les mécanismes de préchargement mais sert ici non pas à charger un bloc dans le cache mais à charger une valeur dans le processeur. Cette valeur ne sera utilisée par d'autres instructions que lorsque la prédiction aura été vérifiée, c'est à dire lorsque l'adresse de la lecture aura été réellement calculée. Ce procédé permet d'améliorer la performance du processeur de manière non négligeable.

Mots-clés : Architecture, processeurs superscalaires, préchargement de données

Abstract

One major restriction to the performance of out-of-order superscalar processors is the latency of loads. We propose to tolerate these latencies by executing loads earlier using a mechanism we called address prediction. The prediction of the address of the data to be loaded is made during the dispatch using the address of the instruction. Thus, loads are executed earlier and, provided the prediction is good, their result might be used as soon as the address is verified. The address prediction technique looks like prefetching techniques but presents a higher gain because the value is directly fetched into the processor rather than in the L1 cache only.

Keywords: Architecture, out-of-order superscalar processors, data preloading

Table des matières

I Prédiction de l'adresse des lectures pour tolérer la latence des accès mémoire 1

1 Introduction 3
- 1.1 Etat de l'art . . . 3
- 1.2 Notre solution . . . 4
- 1.3 Plan du rapport . . . 4

2 Modèle d'exécution - Programmes de test 5
- 2.1 Méthodologie de simulation . . . 7
- 2.2 Programmes de test . . . 7

3 Le mécanisme de prédiction d'adresse 9
- 3.1 La table de prédiction . . . 9
- 3.2 Prédiction d'une lecture . . . 10
- 3.3 Accès à la mémoire . . . 10
- 3.4 Mise à jour de la table . . . 11

4 Résultats 12

5 Conclusion et perspectives 15

II Prediction of addresses for speculating loads 17

1 Introduction 19
- 1.1 Related work . . . 19
- 1.2 Our scheme . . . 20
- 1.3 Organization of the paper . . . 20

2 Baseline Execution Model 21

3 Simulation methodology 23

4 Benchmarks 24

5 The address Prediction Mechanism 26
- 5.1 The Prediction Table . . . 26
- 5.2 Prediction of a load . . . 27
- 5.3 Access to the Memory . . . 27

5.4 Update of the table 27

6 Results **29**

7 Conclusion **32**

References **35**

Première partie

Prédiction de l'adresse des lectures pour tolérer la latence des accès mémoire

Chapitre 1

Introduction

Les processeurs superscalaires à exécution non ordonnée tirent parti de la possibilité d'exécuter les instructions dans un ordre différent de celui du programme pour masquer la latence de certains accès à la mémoire par l'exécution d'autres instructions. Ce n'est toutefois pas suffisant pour masquer la latence de certaines lectures qui nécessitent un accès au cache de deuxième niveau ou –pire– à la mémoire principale [4].

Cette étude se propose de fournir un mécanisme afin d'exécuter les lectures en mémoire plus tôt. L'exécution d'une lecture peut être vue comme deux opérations indépendantes mais ordonnées : le calcul de l'adresse et ensuite l'accès à la mémoire. Ces deux opérations peuvent être réalisées par deux unités fonctionnelles différentes comme dans le microprocesseur Hal [14] ou par une seule unité fonctionnelle qui calcule les adresses et possède une file pour les accès à la hiérarchie mémoire. Dans les deux cas, ces deux étages peuvent être découplés.

1.1 Etat de l'art

Plusieurs solutions ont été avancées pour améliorer l'efficacité des lectures, toutes cherchant à diminuer la latence de ces opérations et certaines d'entre-elles ayant été implémentées dans les microprocesseurs récents. Si le protocole de consistance mémoire est l'ordonnancement faible (ce qui est de plus en plus le cas), les écritures doivent toujours être faites dans l'ordre mais les lectures peuvent être exécutées dans le désordre et dépasser des écritures du moment que les emplacements en mémoire concernés ne sont pas les mêmes et qu'il n'y a pas de dépendances bien sûr. Des techniques ont été proposées pour exécuter spéculativement des lectures en pariant sur le fait qu'il n'y aurait pas d'alias avec les écritures en attente et dont l'adresse n'est pas encore connue [5][6][8]. On appelle cette technique "désambiguation mémoire". Les lectures sont donc exécutées même si l'adresse des écritures précédentes ne sont pas connues mais la donnée lue attend dans la station de réservation que les adresses des écritures précédentes soient toutes connues avant de la propager aux instructions qui l'attendent. Cette technique donne un petit gain de performance et a été considéré pour notre modèle de base décrit plus loin.

Une autre approche pour réduire la latence est le calcul d'adresse rapide [1][2] qui consiste à prédire l'adresse avec un système matériel très simple et donc très rapide. Ainsi, la latence des accès au cache est réduite d'un cycle. Cette solution est donc peu efficace pour les accès au cache secondaire et encore moins pour les accès à la mémoire.

Enfin, la prédiction de valeur [16] prédit la valeur qui va être chargée depuis la mémoire. Elle laisse la donnée prédite dans la station de réservation jusqu'à ce que la prédiction soit vérifiée en accédant à la mémoire. Mais, cela ne présenterait aucun gain si elle n'était également envoyée aux instructions directement dépendantes qui peuvent être exécutées spéculativement et annulées (avant l'envoi du résultat) si la prédiction était mauvaise. Les instructions directement dépendantes peuvent donc être bloquées si leur temps d'exécution est plus court que le temps d'accès. Cette technique est basée sur le fait que pour une instruction de lecture donnée, ses différentes exécutions produisent souvent le même résultat ou du moins un ensemble restreint de valeurs.

Enfin, de nombreuses études ont été menées par le passé sur le préchargement de données dans les caches de données mais il semble que le préchargement ne soit pas une bonne solution pour les processeurs superscalaires car la bande passante entre le cache L1 et le cache L2 est limitée.

1.2 Notre solution

Notre solution a des points communs avec la prédiction de valeur et le préchargement de données. Nous proposons de prédire l'adresse des lectures lorsque les instructions de lectures sont envoyées dans les stations de réservation de manière à accéder à la mémoire même si l'adresse n'est pas calculée (cad si le sopérandes ne sont pas connus). Comme la prédiction de branchement permet de dépasser les dépendances de contrôle, notre technique permet de dépasser les dépendances vraies des lectures. Ainsi, les accès à la mémoire sont effectués plus tôt¿

1.3 Plan du rapport

En premier lieu, nous présentons notre modèle d'exécution ainsi que les programmes tests, puis nous explicitons le mécanisme de prédiction d'adresse et présentons ses résultats en termes de qualité de prédiction. Nous montrons ensuite le gain en performance du processeur apporté par notre dispositif et le même dispositif mais avec une prédiction parfaite.

Chapitre 2

Modèle d'exécution - Programmes de test

L'architecture modélisée met en œuvre une politique d'exécution des instructions non ordonnée et spéculative avec gestion précise des exceptions. Dans ce modèle, après avoir été chargées (les branchements sont toujours correctement prédits) et décodées, les instructions sont émises dans le tampon d'amorçage et insérées dans le tampon de réordonnancement. Le nombre maximum d'instructions pouvant être émises par cycle est appelé le degré du processeur. Nous considérons ici un degré de 8, ce qui correspond probablement à la prochaine génération de processeurs. Les instructions en attente de leurs opérandes dans le tampon d'amorcage ne bloquent pas le décodage : les opérandes manquants vont être envoyés au tampon d'amorçage dès qu'ils sont produits par les instructions précédentes. Les entrées du tampon d'amorçage sont similaires aux stations de réservation de l'IBM360/91 [17] excepté le fait qu'elles sont reliées à plus d'une unité fonctionnelle. Les instructions attendent leurs opérandes dans ces entrées en utilisant une étiquette pour les détecter lorsqu'elles passent sur le bus. Quand les opérandes sont prêts, l'instruction peut être lancée vers une unité fonctionnelle libre et l'entrée est libérée. Les entrées lancées sont sélectionnées parmi celles qui ont leurs opérandes prêts en favorisant les plus anciennes. Quand une instruction a été exécutée, le résultat est renvoyé via un bus à toutes les entrées du tampon d'amorçage. Dans notre modèle, le tampon d'amorçage est divisé en 4 parties : une pour les instructions entières, une pour les instructions flottantes, une pour les lectures/écritures et une pour les branchements. La taille de ces différentes parties est donnée dans le tableau 2.1 et a été défini comme optimale dans [11].

Les instructions sont retirées du tampon de réordonnancement dans l'ordre du programme et les registres sont mis à jour à cette occasion. Cela garantit un

Entier	Flottant	Lecture/écriture	Branchement
48	44	36	24

TAB. 2.1 - *Configuration du tampon d'amorçage*

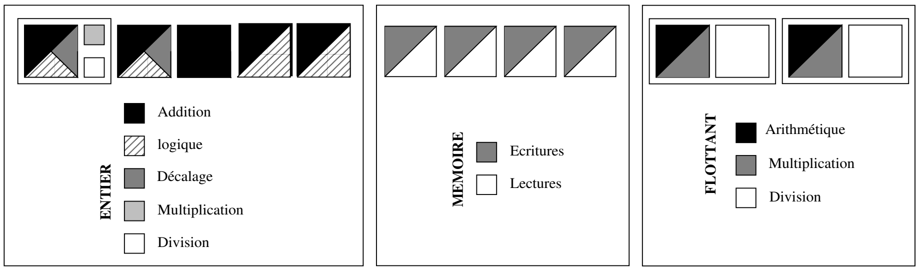

FIG. 2.1 - *Configuration des unités fonctionnelles*

résultat correct et la possibilité de récupérer un état correct en cas de mauvaise prédiction ou d'exceptions. La taille du tampon de réordonnancement est de 96 comme défini dans [10].

En ce qui concerne les unités fonctionnelles, nous avons aussi choisi la configuration considérée comme optimale dans [10]. Celle-ci est brièvement décrite figure 2.1.

Quatre unités fonctionnelles traitent les lectures et les écritures, nous avons modélisé le mécanisme de détection des synonymes proposé dans [1] qui permet d'effectuer une lecture avant une écriture même si l'adresse de celle-ci n'est pas connue en prédisant qu'il ne s'agit pas d'un synonyme (voir la section 1.1).

La hiérarchie mémoire est composée de deux niveaux de caches. Le premier niveau comprend deux caches séparés, un pour les instructions et un pour les données tandis que le deuxième niveau comporte un seul cache. Le cache de données de premier niveau est de 32 Kilo-octets. Il est à correspondance directe, à réécriture et à écriture allouée avec chargement sur écriture [13]. Le lecteur peu familier trouvera une excellente description de son fonctionnement dans [7]. Il supporte quatre accès simultanés depuis le processeur grâce à un entrelaçage sur quatre bancs et est non bloquant [9]. On suppose qu'il peut supporter un nombre infini de requêtes en attente (nous avons mesuré que restreindre à seulement deux requêtes en attente était suffisant pour obtenir une performance sensiblement identique). En cas de succès de l'accès dans ce cache, le temps d'accès est de 1 cycle.

Le cache d'instructions de premier niveau est supposé infini, ce qui influe peu sur les résultats car les programmes utilisés sont petits et seraient contenus dans un cache de 32 Kilo-octets.

Le cache de deuxième niveau est un cache à écriture différée mono-port. Sa taille est de 256 Kilo-octets (taille volontairement limitée car les programmes utilisés ont des ensembles de données petits). Il est associatif par ensembles de 4 blocs. Le temps d'accès à ce cache est de 4 cycles en cas de succès. Il est aussi non bloquant mais ne supporte qu'une requête en attente ce qui est suffisant pour obtenir toute la performance.

La mémoire principale se divise en 8 bancs de 1 Méga-octets. Son temps d'accès représente 20 cycles du processeur. L'interface entre le cache de deuxième niveau et la mémoire principale est un bus à transactions éclatées dont le débit est de 64 bits/cycle.

	Commandes	Instructions	% de L/E	% de L
Espresso	espresso cps.in	26 372 443	30,00	27,24
Compress	compress in	23 522 410	34,01	23,27
Tomcatv	tomcatv	29 912 073	26,65	18,68
Alvinn	backprop	32 154 712	24,88	13,87
Ear	ear if=ref.m22	23 846 933	33,55	20,75
Swm256	swm256 < swm256.in	18 755 032	42,66	33,10
Su2cor	su2cor < su2cor.in	18 458 425	43,34	33,74
Hydro2d	hydro2d < hydro2d.in	22 613 656	35,37	25,81

TAB. 2.2 - *Caractéristiques des programmes simulés*

2.1 Méthodologie de simulation

Le simulateur que nous avons développé est un simulateur dirigé par programme et non par traces. Il exécute du code objet Motorola 88k produit par le compilateur mcc88k de Microtec Research avec l'optimiseur actif. Les bibliothèques ont été partiellement réécrites excepté les appels au système opératoire qui sont émulés (l'effet produit sur les valeurs des registres par les appels est conservé) et ne sont donc pas exécutés par le processeur. Ce type de simulation permet de vérifier la validité de notre modèle en exécutant tout le programme de test et en vérifiant que les résultats produits sont corrects.

De manière à raccourcir les temps de simulation, pour chaque programme de test le processeur exécute 4 millions d'accès mémoire et les mesures sont effectuées pour les 8 millions d'accès suivants. Ainsi, nous évitons les effets d'un démarrage à froid sur les caches et sur notre système de prédiction.

2.2 Programmes de test

Les programmes utilisés sont issus de la suite SPEC92. Afin de pouvoir évaluer l'influence de la hiérarchie mémoire, 8 programmes ont été retenus parmi ceux qui présentent le plus grand nombre de données : *espresso, compress, tomcatv, alvinn, ear, swm256, su2cor, hydro2d*. Leurs caractéristiques sont données dans la table 2.2. Comme nous sommes principalement concernés par les accès mémoire, nous donnons ici, le nombre d'instructions simulées, le pourcentage d'instructions qui sont des lectures ou écritures (L/E) et le pourcentage de lectures (L). Nous avons utilisé les jeux d'entrée standards donnés par SPEC.

La figure 2.2 montre le nombre moyen d'instructions exécutées par cycle pour notre modèle de base, c'est à dire sans prédiction d'adresse.

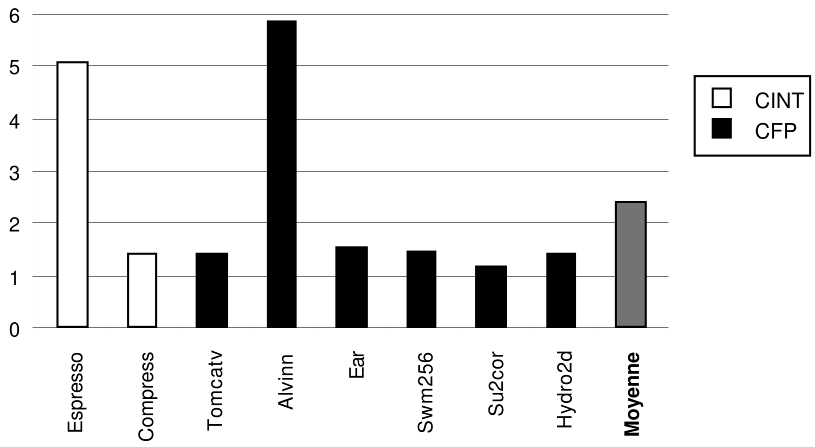

FIG. 2.2 - *Instructions par cycle pour le modèle de base*

Chapitre 3

Le mécanisme de prédiction d'adresse

Les lectures peuvent être divisées en 3 catégories :

1. les lectures à pas nul. Elles accèdent toujours à la même adresse.
2. Les lectures à pas constant. Elles accèdent en mémoire à des adresses incrémentées (ou décrémentées) régulièrement.
3. Les lectures à pas irréguliers.

La figure 3.1 montre la répartition de ces lectures lors de l'exécution des programmes.

Comme la prédiction de branchement, la prédiction d'adresse déduit le comportement des instructions à partir des exécutions précédentes. A chaque cycle, huit instructions sont chargées et donc jusqu'à huit instructions peuvent être prédites en parallèle.

Quand la prédiction d'adresse est correcte, le gain en performance provient du fait que l'accès peut débuter avant que les opérandes soient prêts. On dépasse ainsi la limite du modèle data-flow. Suivant l'endroit où se trouve la donnée (cache de premier niveau, de deuxième niveau, mémoire) et le temps entre le moment où l'instruction est lancée et le moment où les opérandes sont prêts, le gain peut varier de 0 à 20 cycles.

Nous considérons ici deux cas extrêmes.

Le cas *parfait* prédit correctement toutes les instructions de lecture. Ce cas nous permet d'évaluer l'accroissement de performance maximum (en nombre d'instructions exécutées par cycle) que peut apporter la prédiction d'adresse à un processeur de type similaire à celui que nous avons modélisé.

Le second cas, appelé *cas simple*, a un coût matériel relativement peu élevé. Il est décrit dans les sections suivantes.

3.1 La table de prédiction

Le mécanisme de prédiction des adresses est largement inspiré de [3]. Il comporte une table de 256 entrées (taille permettant d'obtenir une bonne qualité de prédiction) qui est accédée en correspondance directe avec l'adresse de

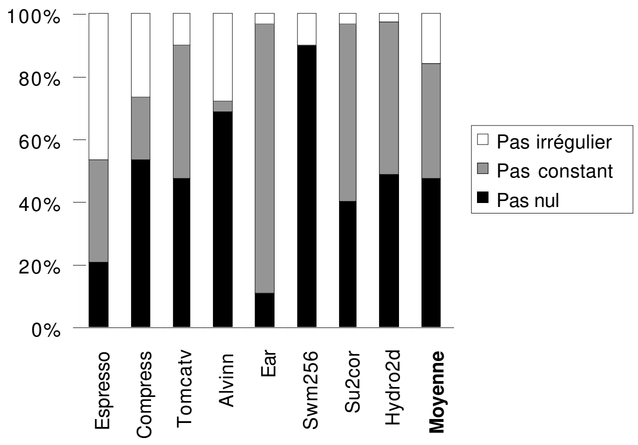

FIG. 3.1 - *Répartition des lectures suivant leur pas*

l'instruction à prédire. Cette table est chargée de mémoriser le comportement des lectures. Pour ce faire, chaque entrée est composée de 5 champs : le champ adresse_instruction permet de vérifier que les informations de l'entrée concerne bien l'instruction considérée. Le champ adresse_accès garde l'adresse du dernier accès effectué par cette instruction. Le champ pas est la différence entre les adresses des deux derniers accès de cette instruction. Le champ état indique si l'on peut effectuer une prédiction ou pas. Le champ avance indique le nombre d'occurrences de cette instruction présentes dans le processeur qui ont été prédites et qui n'ont pas encore mis à jour la table.

3.2 Prédiction d'une lecture

Avant l'émission d'une instruction dans une entrée du tampon d'amorçage, si l'entrée correspondante de la table de prédiction contient des données relatives à cette instruction (c'est à dire si le champ adresse_instruction est égal à l'adresse de l'instruction), et si la prédiction est autorisée (état *Stable*), l'avance est incrémentée et l'adresse prédite est $adresse\_acc\grave{e}s + (pas \times avance)$. Cette adresse prédite est placée dans l'entrée du tampon d'amorçage en même temps que l'instruction et un bit dans l'entrée est positionné pour indiquer que l'adresse n'a pas été calculée mais seulement prédite.

3.3 Accès à la mémoire

Lorsqu'une lecture est sélectionnée pour être exécutée, si ses opérandes sont prêts l'adresse est calculée et utilisée pour l'accès à la hiérarchie mémoire. Si les opérandes ne sont pas prêts et si l'adresse a été prédite, l'adresse prédite est utilisée pour l'accès. Une fois la valeur lue en mémoire, celle-ci est gardée dans l'entrée du tampon d'amorçage jusqu'à ce que les opérandes soient calculés. Si l'adresse calculée est égale à l'adresse prédite, la valeur est transmise aux entrées du tampon d'amorçage qui l'attendent sinon l'accès est relancé avec l'adresse correcte.

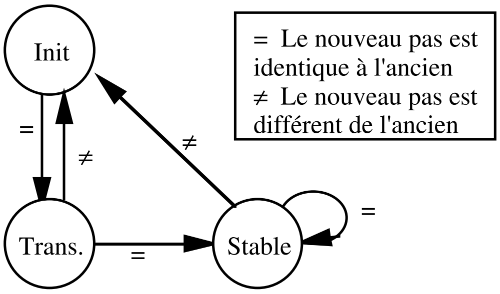

FIG. 3.2 - *Automate à états du prédicteur*

3.4 Mise à jour de la table

Lors de la mise à jour, si l'instruction n'est pas présente dans la table, elle y est placée avec état=*Init*, avance=1 et pas=0. Sinon, le champ pas est mis à jour avec la différence entre l'adresse présente dans la table et celle qui est accédée, le champ état évolue suivant l'automate à états de la figure 3.2, le champ avance est décrémenté s'il est différent de 0 et l'adresse accédée est placée dans le champ adresse_accès.

Deux méthodes de mise à jour ont été évaluées. Soit la table est mise à jour dans l'ordre du programme c'est à dire lorsque les instructions sont retirées du tampon de réordonnancement. La mise à jour est alors assez tardive et la prédiction est fortement anticipée par rapport à l'information présente dans la table car les boucles ne sont généralement pas longues et plusieurs itérations d'une boucle peuvent être présentes dans le processeur. Soit la table est mise à jour le plus tôt possible, c'est à dire dès que l'adresse de l'accès est calculée. La table est alors mise à jour dans le désordre, ce qui peut nuire à la qualité de la prédiction. Toutefois, la politique d'amorçage des instructions est d'exécuter l'instruction la plus ancienne, ce qui favorise l'exécution dans l'ordre. Il n'est donc pas étonnant que nous n'ayons pas observé, en moyenne, de différence notable entre ces deux méthodes.

Chapitre 4

Résultats

La figure 4.1 montre les améliorations de performance relatives à la performance du modèle de base montrée sur la figure 2.2. Comme dans le cas *parfait* et dans le cas *simple*, la valeur renvoyée attend dans l'entrée du tampon d'amorçage que les opérandes soient calculés, la seule différence entre ces deux cas est la qualité de prédiction, tous les autres paramètres de l'architecture restant inchangés.

La figure 4.1 montre que *espresso* et *compress* sont les programmes qui accusent la plus grande différence entre les deux cas. La figure 4.2 explique cette différence pour *espresso* qui a une qualité de prédiction de seulement 63%. ceci est dû au fait que presque la moitié des lectures ont un pas irrégulier (voir figure 3.1).

Cependant, la qualité de prédiction n'est pas la seule explication puisque *compress* présente déjà une bonne qualité de prédiction dans le cas *simple*. De plus, *ear* a aussi une mauvaise prédiction et la performance n'est quasiment pas améliorée dans le cas *parfait*.

Voyons ce qui se passe au niveau de la hiérarchie mémoire. La figure 4.3 montre l'accroissement des taux d'échecs dus à la prédiction d'adresse. Le taux d'échecs du cache de premier niveau est très sensible à la qualité de prédiction pour *espresso* et ceci peut expliquer la différence entre les améliorations de performance des cas *simple* et *parfait*. En ce qui concerne *compress*, le taux d'échecs du cache de premier niveau croît de 10 à 8% pour les cas *simple* et *parfait*. Cela

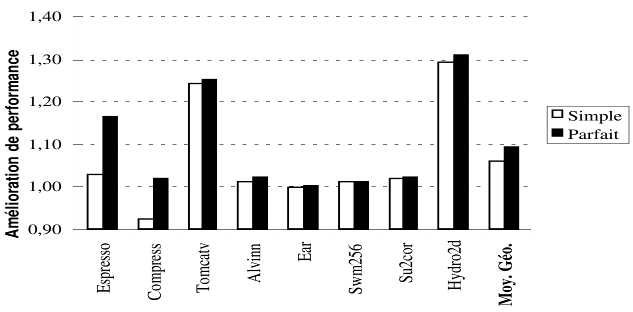

FIG. 4.1 - *Amélioration de performance*

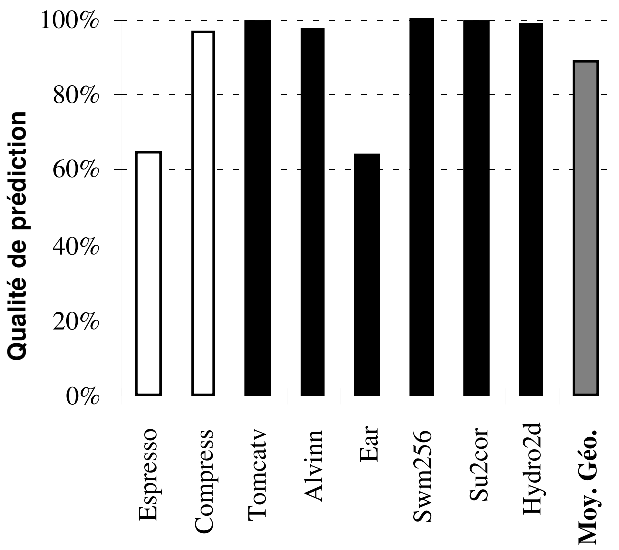

FIG. 4.2 - *Qualité de prédiction*

peut expliquer les mauvaises améliorations de performance observées pour ce programme si la latence des lectures en échec ne peut pas être compensée par l'exécution d'instructions indépendantes. En ce qui concerne *ear*, la prédiction parfaite supprime l'accroissement du taux d'échecs du cache de premier niveau mais n'améliore pas la performance. De fait, les études sur les techniques de préchargement avaient déja montré que ces techniques ont peu d'effet sur ce programme. Ceci est probablement dû au fait que *ear* a moins de 5% de pas nuls, les pas réguliers devant être mal répartis.

La figure 4.4 montre le pourcentage de lectures qui sont exécutées avec une adresse incorrecte et avec une adresse correctement prédite. En effet, il est, par exemple, inutile de prédire très bien si cela ne sert jamais! Cette figure montre clairement que les programmes qui ont une mauvaise qualité de prédiction (*espresso* et *ear*) ont plus de lectures exécutées avec une adresse mal prédite que de lectures exécutées avec une adresse correctement prédite bien que la qualité de prédiction soit supérieure à 50%. De plus, *ear* présente la plus grande différence entre ces deux valeurs. Enfin, les programmes qui ont les meilleures améliorations de performance sont ceux qui ont le plus grand nombre de lectures exécutées avec une adresse correctement prédites. ils présentent également peu de lectures exécutées avec une mauvaise adresse.

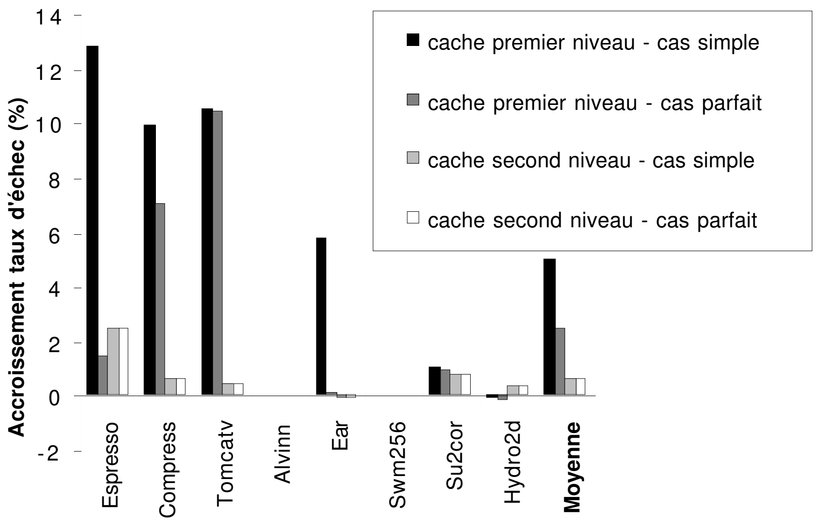

FIG. 4.3 - *Accroissement du taux d'échecs des caches de premier et de second niveau*

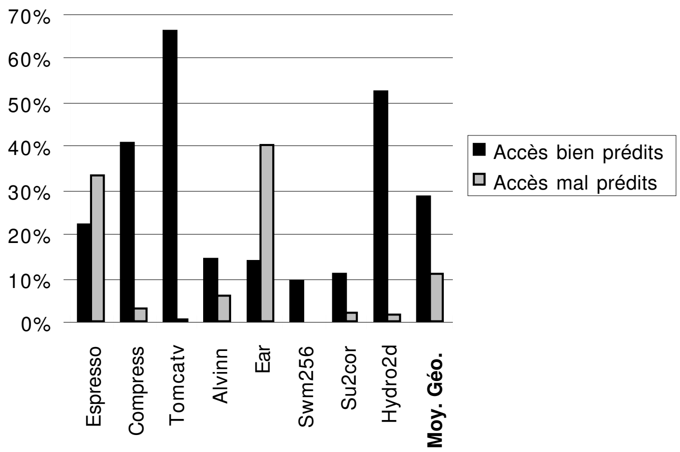

FIG. 4.4 - *Pourcentages de lectures exécutées avec une adresse prédite*

Chapitre 5

Conclusion et perspectives

Ce mécanisme ressemble aux mécanismes classiques de préchargement mais il diffère sur plusieurs points.

Les mécanismes de préchargement se contentent de charger un bloc dans le cache de premier niveau alors que notre dispositif de prédiction d'adresse charge directement une donnée dans le processeur. C'est un point important car la bande passante entre les caches de premier et second niveau est limitée et constitue un goulet d'étranglement [15].

Le préchargement engendre plus de requêtes inutiles car il prédit à la fois l'occurence d'une instruction de lecture et son adresse. Dans notre cas, on est certain qu'une instruction de lecture va être exécutée. Les requêtes au cache sont donc toujours justifiées, seule l'adresse peut être erronée.

Le préchargement charge un bloc pour la prochaine occurence de l'instruction. A priori, il lance donc une requête plus tôt que notre système. Mais le plus souvent, deux occurences successives d'une instruction de lecture sont présentes dans le processeur et ont donc également été prédites par notre système.

La prédiction d'adresse parfaite montre que cette approche permet d'améliorer les performances du processeur d'environ 10%. Le cas simple étudié ici apporte déjà une amélioration notable (plus de 5%). Il reste de nombreuses alternatives à étudier et à évaluer. L'automate qui définit si une lecture doit être prédite ou non privilégie la qualité de la prédiction au détriment du nombre de prédictions de manière à limiter les accès inutiles aux caches. Une solution pour limiter encore plus ces accès aurait un très bon impact sur l'amélioration de performance.

Part II

Prediction of addresses for speculating loads

Chapter 1

Introduction

The aim of out-of-order superscalar processors is to implement a restricted data-flow model. In this model, instructions are executed when their operands are ready and when a functional unit is free. Data flow is restricted by the size of the instruction window which represents the number of instructions that are in the processor. Thus, parallelism is limited only by true dependencies and by resource limitations. However, some instructions, especially loads, may have such long latencies that the instruction window can be full of true-dependent instructions. In this case, the processor is stalled until the load is resolved.

The challenge is to provide a mechanism for executing loads earlier. Load execution may be seen as two independent but ordered operations: first, the calculus of the address and second, the access to the memory hierarchy. These two stages may be processed by two different functional units with their own reservation stations as in the Hal microprocessor [14], or by a single functional unit that computes the address followed by a queue for the access to the memory hierarchy. In any case, these two stages may be decoupled.

1.1 Related work

In order to improve the efficiency of loads, several schemes have been proposed and most of them have been implemented in commercial processors. In weak ordering, stores must be done in order but loads may be executed out-of-order and can bypass stores if they are not at the same address. Often, several stores are waiting in reservation stations for their operands being calculated and, thus cannot be bypassed since their addresses are not known. Some work has been done on techniques for speculating on the absence of alias with previous stores in order to execute loads earlier[5][6][8] and is refered as memory disambiguation. Loads are executed even if some preceding stores are not known but the loaded data waits in the reservation station until addresses of all preceding stores are known. This scheme gives a little performance gain and has been considered in our baseline execution model.

Another approach to reduce latency of loads is fast address calculation[1][2] which consists in predicting the address with a very simple harware scheme. Thus, if the data resides in the cache, it results in a zero-cycle load. This scheme reduces the latency of loads by one cycle. So, it is not effective for long-latency loads.

Value prediction [16] predicts the value that will be loaded from the memory. It leaves the predicted load in the reservation station until this prediction is verified by accessing to the memory hierarchy. However, the predicted value is forwarded to directly-dependent instructions that may be speculatively executed and cancelled if the prediction was wrong. This scheme is based on the fact that, for a given static load, its executions result in a small amount of different values.

Finally, many studies have been conducted in the past on data prefetching in data caches but it seems that prefetching is not efficient for superscalar processors because the bandwidth between the L1 and L2 caches is limited.

1.2 Our scheme

We propose to predict the data address of loads when they are dispatched in order to access to the memory hierarchy even if the address cannot be calculated (i.e. if the operands are not known). As well as branch prediction bypasses control dependences of branches, our scheme bypasses true dependences of loads. Thus, these loads access to the memory hierarchy earlier.

1.3 Organization of the paper

This paper is organized as follows: the overall execution model and the benchmarks used for performance evaluation are presented in the next section. Then we detail a simple scheme for address prediction and show its accuracy. Finally, we present the speedups obtained with this scheme and with a perfect-prediction scheme.

Chapter 2

Baseline Execution Model

The modeled architecture implements an out-of-order execution-issue policy and speculative execution to best exploit instruction-level parallelism.

In this model, after being fetched (branches are always well-predicted) and decoded, instructions are dispatched in the instruction-issue buffer and enqueued in the reorder buffer. The upper bound of the number of instructions dispatched each cycle is called the degree of the processor. In this paper, we will consider an aggressive degree of 8, which probably corresponds to the next generation of processors. Instructions waiting for their operands do not stall the decoding: the missing operands will be forwarded to the instruction-issue buffer as soon as they are produced by previous instructions. The entries of the instruction-issue buffer are similar to the entries of the reservations stations of the IBM360/91 [17] except that they are linked to more than one functional unit. Instructions are waiting in these entries for their operands using a tag to retrieve them. When the operands are ready, the instruction may be issued to a free functional unit and the entry is resetted. Entries are selected among all those that have their operands ready according to the program order. When an instruction has been executed, the results are forwarded to all entries of the instruction-issue buffer. In our model, the instruction-issue buffer is divided into four parts: one for integer instructions, one for floating-point instructions, one for loads and stores and one for branches. The sizes of these parts are given in table 2.1 and have been defined as the optimal sizes for our model in [11].

Instructions are retired (dequeued from the reorder buffer) in the order of the program and the registers are updated at that time. This guarantees the correct result and the possiblity of recovering from mispredictions of branches and from exceptions. The size of the reorder buffer is 96 as defined in [10].

Concerning the functional units, we also choose the configuration described as optimal in [10] which is briefly recalled in figure 2.1.

Concerning load/stores, we have four functional units executing either loads

Integer	Floating-Point	Load/Store	Branch
48	44	36	24

TAB. 2.1 - *Configuration of the instruction-issue buffer*

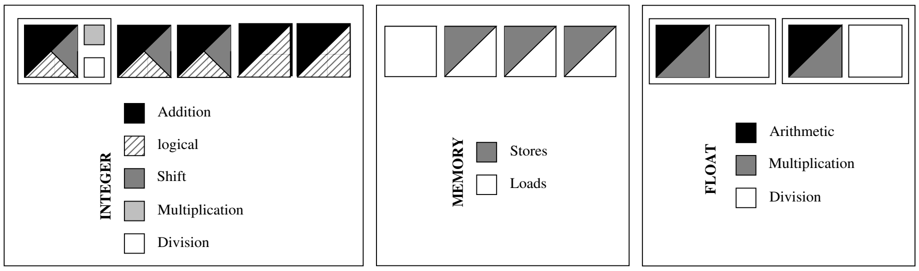

FIG. 2.1 - *Configuration of the functional units*

or stores and the memory disambiguator presented in section 1.1 is implemented.

The memory hierarchy includes a L1 32-KB data cache. This cache is composed of 4 interleaved banks that allow four simultaneous accesses by the processor (one access per L/S functional unit) provided there is no bank conflict. It is a write-back direct-mapped cache with fetch-on-write and write-allocate policies[13]. As most of caches today, it is also non-blocking [9]. We suppose that it can support an infinite number of pending loads since measures show that restricting the number fo pending loads to 2 gives the same performance. The hit access time is 1 cycle. The L1 instruction cache is supposed to be infinite since the code of the benchmarks we consider is small and can be stored almost entirely in a 32-KB cache.

The second level cache is also a write-back, non-blocking cache but it only supports one pending access. We measured that it is sufficient to obtain almost all the possible performance. The hit access time is 4 cycles. The L2 cache size has intentionally been limited to 256 KB in order to have a reasonable amount of misses since the benchmarks we used are konwn to have small data sets. It is a 4-way associative cache.

The main memory is divided in eight 1-MB banks. Its access time is 20 processor cycles and the bus is a split-transaction one. Its width is 64 bits.

Chapter 3

Simulation methodology

The simulator we have developped is a full-execution one and executes Motorola 88k code produced by the Microtec Research mcc88k compiler. All optimizations were turned on when compiling the benchmarks. Libraries have been partially rewritten and are simulated except the system calls which are emulated. Thus, they are not executed by our processor model but their effects on the register values are kept in order to obtain the good results. Full-execution allows us to verify the validity of our model by executing the whole benchmark and verifying that the produced results are correct.

In order to shorten the simulation time, for each benchmark, the processor executes four millions memory accesses and measurements are made for the eight next millions of memory accesses. Thus, we avoid the effects of a cold start on the caches and on our prediction mechanism.

Chapter 4

Benchmarks

The benchmarks used belong to the SPEC92 suite and are among those which have the larger data sets in order to observe the impact of our scheme on the performance of the memory hierarchy. Their characteristics, measured between four and eight millions of memory accesses, are given in table 4.1. As we are mainly concerned by memory accesses, we report on the number of instructions simulated, the percentage of instructions that are load/stores and the percentage of loads. We used the standard input sets.

Figure 4.1 shows the mean number of instructions per cycle for each benchmark for our baseline execution model, i.e. without any load prediction.

	Command	Instructions	% of L/S	% of loads
Espresso	espresso cps.in	26,372,443	30.00	27.24
Compress	compress in	23,522,410	34.01	23.27
Tomcatv	tomcatv	29,912,073	26.65	18.68
Alvinn	backprop	32,154,712	24.88	13.87
Ear	ear if=ref.m22	23,846,933	33.55	20.75
Swm256	swm256 < swm256.in	18,755,032	42.66	33.10
Su2cor	su2cor < su2cor.in	18,458,425	43.34	33.74
Hydro2d	hydro2d < hydro2d.in	22,613,656	35.37	25.81

Tab. 4.1 - *Characteristics of the simulated benchmarks*

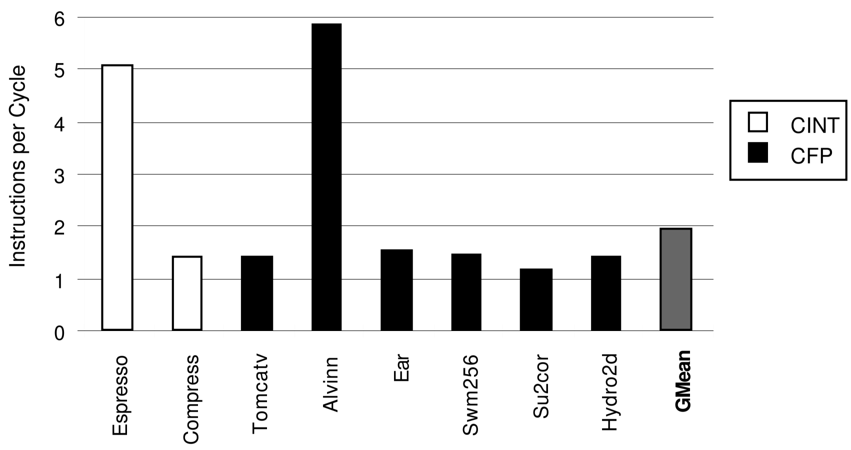

FIG. 4.1 - *Instructions per cycle for the baseline model*

Chapter 5

The address Prediction Mechanism

Dynamic loads can be divided into three categories:

1. Zero-stride loads: two consecutive occurences of a load that access to the same address.

2. Constant-stride loads: consecutive occurences of a load that access to locations the address of which is increased (or decreased) regularly.

3. Irregular stride loads.

Figure 5.1 shows the distribution of these dynamic loads in our benchmarks.

As for branch prediction, address prediction snoops earlier executions of loads to deduce their behaviour. Each cycle, up to eight instructions are dispatched and up to eight can be predicted in parallel.

When the address prediction is good, the performance gain is due to the fact that the access may start before the operands are ready, thus breaking the data-flow limit. Depending on where the data is (L1, L2, memory) and the time between the issue of the predicted load and the cycle where the operands are ready, the gain will vary between 1 and 20 cycles.

We consider two extreme cases. The *perfect* case predicts correctly all loads. This gives an upper bound to the speedup that can be obtained with our scheme.

The second case is a *simple* case which has a relatively small hardware cost. It is described in the next subsections.

5.1 The Prediction Table

The prediction mechanism is largely inspired from [3]. It consists in a direct-mapped 256-entry table that is indexed by the address of the dispatched instruction. This table keeps the behaviour of loads. Each entry has 5 fields: `instruction_address` is the address tag of the instruction, `access_address` keeps the last address accessed by this load, `stride` is the stride between the two last accesses of the load, `times` is the number of occurences of this load that have been predicted but have not yet updated the table, and `state` indicates if the prediction can be done.

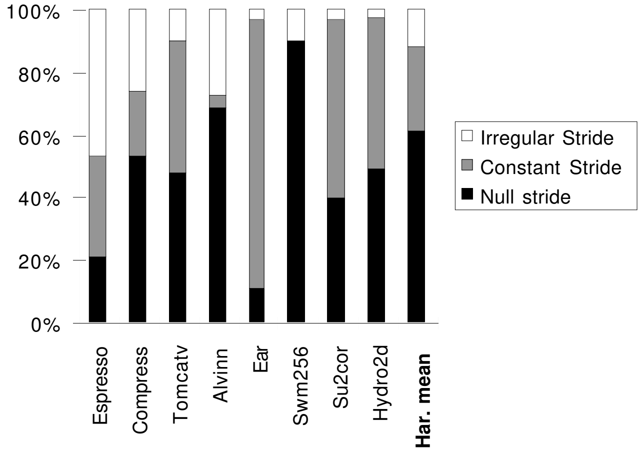

FIG. 5.1 - *Distribution of the loads according to their stride*

5.2 Prediction of a load

Before the dispatch of a load in an entry of the instruction-issue buffer, the table is accessed with the address of the load. If the table entry concerns this load (ie if instruction_address is the address of the load being issued), the prediction is made if state is *Steady*. The predicted address is $access\_address + (stride \times times)$ and times is incremented. The predicted address is placed in the instruction-issue buffer entry with the instruction and a bit in the instruction-issue buffer entry indicates that the address has not been calculated but only predicted.

5.3 Access to the Memory

When a load is selected for being executed, if its operands are ready, the address is calculated and used for the access to the memory hierarchy. If not and if the address has been predicted, this predicted address is used for the access. When they are returned by the memory hierarchy, the accessed data are kept in the instruction-issue buffer entry until the operands are ready and the address has been calculated. If the calculated address equals the predicted address, the data are forwarded to the instruction-issue buffer entries that are waiting for them. Otherwise, a new access to the cache with the correct address is issued.

5.4 Update of the table

The prediction table may be updated in program order i.e. when the instructions are retired. The prediction may suffer from this because the instruction window is large and several occurences of the load may be in the processor, their addresses calculated and the table not updated. This means that the table does not reflect the latest information about the load behaviour.

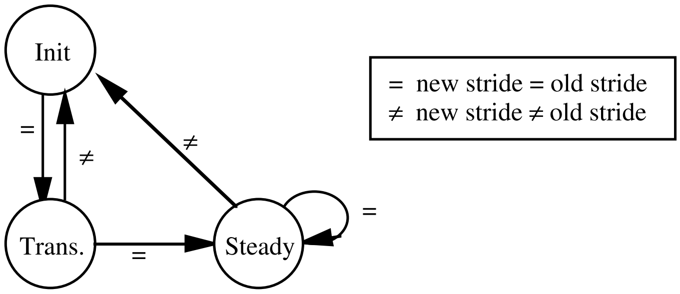

FIG. 5.2 - *State automata*

The prediction table may also be updated as soon as the addresses are calculated. In this case, the update is not in program order but it is made earlier. As the instructions selected for execution are the oldest ones among those that have their address calculated , the program order is nonetheless partially respected .

This is why we did not observe a noticeable difference between these two updating methods.

When updating the table, either the load is not in the table and a new entry is created in the *Init* state, or the load is in the table and the new stride is calculated. State is updated according to the automata shown in figure 5.2. Times is decremented if not equal to zero. The address of the access is placed in access_address.

When the address prediction is good, the performance gain is due to the fact that the access may start before the operands are ready, thus breaking the data-flow limit. Depending on where the data is (L1, L2, memory) and the time between the issue of the predicted load and the cycle where the operands are ready, the gain will vary between 1 and 20 cycles.

Chapter 6

Results

Figure 6.1 shows the speedups (relative to the baseline results shown in figure 4.1) obtained when all loads are well-predicted (perfect) and with our simple prediction scheme describe in the preceding section. Note that in the two cases, speculatively loaded data wait in the instruction-issue buffer entry until the address is verified. Thus, the difference between the simple and perfect cases is due only to the accuracy of the prediction which is 100% for the perfect case. The accuracy of the simple case is given in figure 6.2 for each benchmark.

This figure shows that *espresso* and *compress* are the benchmarks that presents the biggest difference according to the prediction accuracy. Figure 6.2 explains this for *espresso* which has a prediction accuracy of only 63% and this is due to the fact that half of the load have an irregular stride (figure 5.1). However, the accuracy is not the only explanation since *compress* already presents a good accuracy in the simple case. Moreover, *ear* has also a small prediction accuracy and does not benefit from the perfect scheme.

Let use see what happens in the memory hierarchy. Figure 6.3 shows the increase of the miss rates due to the address prediction. The L1 miss rate for *espresso* is highly sensitive to the prediction accuracy and this might explain the difference in the speedups. Concerning *compress*, for both simple and perfect cases, the number of L1 misses is increased of 10 to 8%. This might explain the bad speedups observed for this benchmark if the latency due to the misses cannot be overlapped with the execution of independent instructions. Concerning *ear*, the perfect prediction suppresses the increase of the L1 misses but it does not increase the speedup. People working on data prefetching has already noticed that this benchmark is not a good candidate for this kind of technique. This is probably due to the fact that ear has less than 5% of null strides. To further investigate this, figure6.4 shows the percentage of all dynamic loads that are well-predicted and executed with the well-predicted address.

This figure also shows the percentage of all dynamic loads that are executed with the mispredicted address. It clearly shows that the benchmarks which have a bad prediction (*espresso* and *ear*) have more loads executed with a mispredicted address than loads executed with a well-predicted address. However, *ear* presents the greatest difference. Finally, the benchmarks that have the best speedups (*tomcatv* and *hydro2d*) are those who have the greatest numbers of loads executed with a well-predicted address and they also presents few loads executed with a mispredicted address.

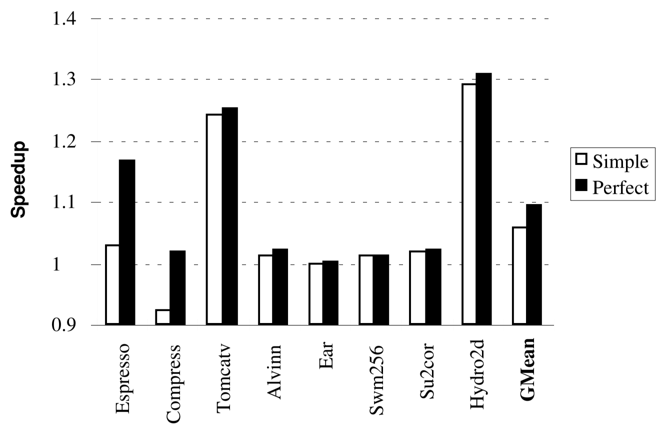

FIG. 6.1 - *Speedups*

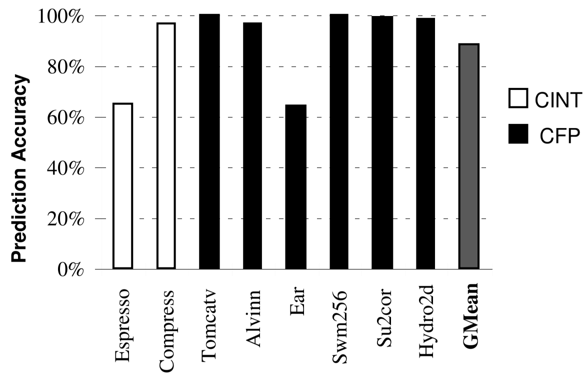

FIG. 6.2 - *Accuracy of the predictions*

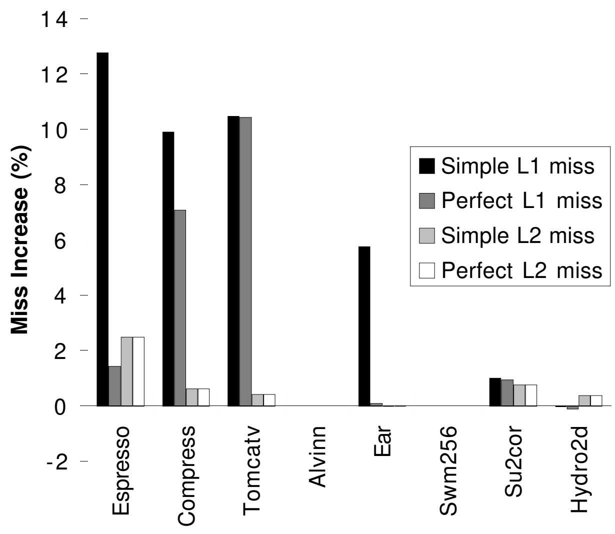

FIG. 6.3 - *L1 and L2 Miss Rate Increase*

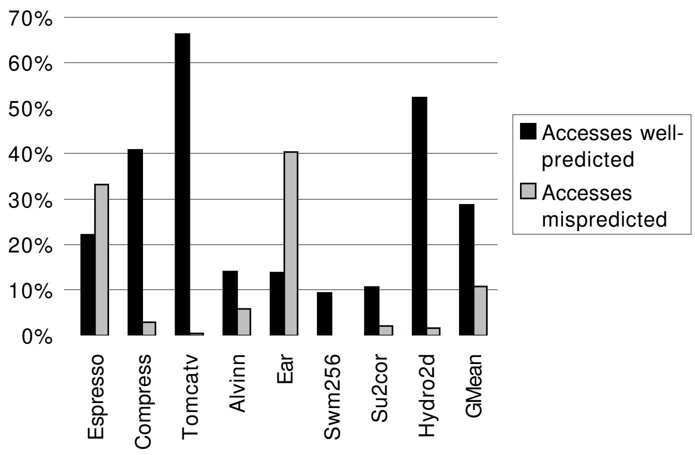

FIG. 6.4 - *Percentages of loads executed with a predicted address*

Chapter 7

Conclusion

This mechanism looks like data cache prefetching schemes. However, it is different in several ways.

The main difference is that address prediction loads data directly in the processor rather than loading a block into the L1 (or L2) cache. This is an essential point since the bandwidth between the L1 and the L2 cache is limited and constitutes a bottleneck as it has been observed in [15].

Address prediction generates less unuseful requests to the L2 cache than cache prefetching. Address prediction predicts the address of a load that *will* be executed when prefetching additionally predicts that this load will be executed in the future.

As cache prefetching loads a data block for the next occurence of a load, one may think that it predicts earlier and, consequently, is able to better tolerate memory latency. This not true since the instruction window of eight-way superscalar processors will be large (96 instructions in our case) and consequently, will contain several occurences of a load. Thus, address prediction also predicts the next occurence of a load before the first one is finished.

The perfect case shows that address prediction is quite an interesting solution since it may increase the performance by almost 10%. Even the simple case shown here offers a 5% improvement of the performance. Moreover, many alternatives are possible either for the address prediction mechanism or for its insertion in the processor execution model. The automata that defines if a load can be predicted privileges the quality of prediction against the number of predictions in order to limit unuseful accesses to the caches. However, a solution to further limit the increase of the L2 miss rate should have a great impact on the speedup.

Bibliographie
References

Bibliographie

[1] T.M. Austin, D.N. Pnevmatikatos, *Streamlining Data Cache Access with Fast Address calculation*, 22nd Annual International Symposium on Computer Architecture, June 1995.

[2] T.M. Austin, G.S. Sohi, *Zero-cycle Loads: Microarchitecture Support for Reducing Load Latency*, 28th Annual International Symposium on Microarchitecture, November 1995.

[3] J-L. Baer, T-F. Chen, *An Effective On-Chip Preloading Scheme To Reduce Data Access Penalty*, Supercomputing'91, November 1991.

[4] D. Burger, J. R. Goodman, A. Kägi, *Quantifying Memory Bandwidth Limitations of Current and Future Microprocessors*, 23ème International Symposium on Computer Architecture, May 1996.

[5] M. Franklin, *The Multiscalar Architecture*, PhD thesis, University of Wisconsin, Madison, 1993.

[6] D.M. Gallagher, W.Y. Chen, S.A. Mahlke, J.C. Gyllenhaal, W-M.W. Hwu, *Dynamic Memory Disambiguation Using the Memory Conflict Buffer*, 6th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1994.

[7] J. L. Hennessy, D. A. Patterson, *Architecture des ordinateurs, une approche quantitative*, International Thomson Publishing, 1996.

[8] A.S. Huang, G. Slavenburg, J.P. Shen, *Speculative Disambiguation: A Compilation Technique for Dynamic Memory Disambiguation*, 21st Annual International Symposium on Computer Architecture, April 1994.

[9] D. Kroft, *Lockup-Free Instruction Fetch/Prefetch Cache Organization*, 8th Annual International Symposium on Computer Architecture, May 1981.

[10] S. Jourdan, P. Sainrat, D. Litaize, *Exploring Configurations of Functional Units in an Out-Of-Order Superscalar Processor*, 25th Annual International Symposium on Computer Architecture, June 1995.

[11] S. Jourdan, P. Sainrat, D. Litaize, *An Investigation of the Performance of Various Instruction-Issue Buffer Topologies*, 28th Annual International Symposium on Microarchitecture, November 1995.

[12] S. Jourdan, P. Sainrat, D. Litaize, *Tampons d'amorçage des microprocesseurs superscalaires à exécution non ordonnée : une analyse*, Renpar'8, May 1996.

[13] N.P. Jouppi, *Cache Write Policies and Performance*, 20th Annual International Symposium on Computer Architecture, June 1993.

[14] M. Simone, A. Essen, A. Krishnamoorthy, T. Maruyam, N. Patkar, M. Ramaswami, M. Shebanow, V. Thirumalaiswamy, D. Tovey, *Implementation Trade-offs in Using a Restricted Data Flow Architecture in a High Performance Microprocessor*, 22nd Annual International Symposium on Computer Architecture, June 1995.

[15] A. Seznec, F. Lloansi, *About Effective Cache Miss Penalty on Out-Of-Order Superscalar Processors*, technical report INRIA 2726, November 1995.

[16] M.H. Lipasti, C.B. Wilkerson, J.P. Shen, *Value Locality and Load Value Prediction*, 7th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1996.

[17] R.M. Tomasulo, *An Efficient Algorithm for Exploiting Multiple Arithmetic Units*, IBM Journal, vol. 11, 1967.